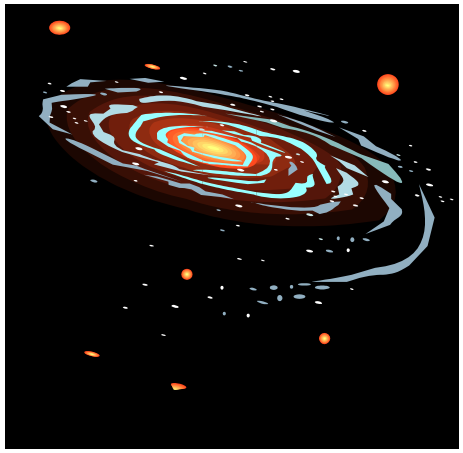
CS 4
Fundamentals of Computer Programming

Lecture 4: January 12, 2015

Space and time complexity



Last time

- Recursion
- How computations unfold:
  - linear recursive processes
  - linear iterative processes
  - tree recursive processes

Today

- Finish description of how processes unfold
- Learn how to quantify the cost of procedures in terms of space and time (asymptotic complexity)

Recall

- Last time we analyzed the behavior of a tree-recursive function to compute fibonacci numbers:

```
(define (fib n)
    (if (< n 2)
        n
        (+ (fib (- n 1))
           (fib (- n 2)) )))
```

- Requires running time which is *exponential* in **`n`**
- Unacceptably large!
- Need alternative…

Alternate method

- Again, count up to get the solution:
- n: 0 1 2 3 4 5 6 7 …
- fib(n): 0 1 0+1=1 1+1=2 2+1=3 …
- fib(n): 0 1 1 2 3 5 8 13 …

Iterative fib

```
(define (ifib n)
  (fib-iter 1 0 n))
```

state variables

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))
```

helper function

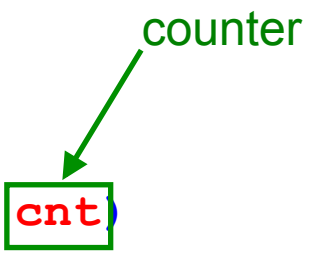
Iterative fib

```
(define (ifib n)
  (fib-iter 1 0 n))

(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))
```

two consecutive fib values

Iterative fib

```
(define (ifib n)
  (fib-iter 1 0 n))

(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))
```



Iterative fib

```
(define (ifib n)
  (fib-iter 1 0 n))

(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))
```

Iterative fib

```
(define (ifib n)
  (fib-iter 1 0 n))

(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))
```

Iterative fib

```
(define (ifib n)
  (fib-iter 1 0 n))

(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))
```

Evolution of iterative `fib`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`
- `(fib-iter 5 3 1)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`
- `(fib-iter 5 3 1)`
- `(fib-iter 8 5 0)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`
- `(fib-iter 5 3 1)`
- `(fib-iter 8 5 0)`
- Answer: 5

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`
- `(fib-iter 5 3 1)`
- `(fib-iter 8 5 0)`

countdown

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`
- `(fib-iter 5 3 1)`
- `(fib-iter 8 5 0)`

stored state

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`
- `(fib-iter 5 3 1)`
- `(fib-iter 8 5 0)`

fibonacci sequence

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

Evolution of iterative `fib`

- `(ifib 5)`
- `(fib-iter 1 0 5)`
- `(fib-iter 1 1 4)`
- `(fib-iter 2 1 3)`
- `(fib-iter 3 2 2)`
- `(fib-iter 5 3 1)`
- `(fib-iter 8 5 0)`

```
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f)
                fnext
                (- cnt 1))))
```

fibonacci sequence

(starting from 1)

Analyzing iterative fib

```
(ifib 5)
(fib-iter 1 0 5)
(fib-iter 1 1 4)
(fib-iter 2 1 3)
(fib-iter 3 2 2)
(fib-iter 5 3 1)
(fib-iter 8 5 0)
5
```

- What kind of process is this?
  - linear iterative
- How many calls to `fib-iter` for input `n`?

  `n+1`
- Space required?

  constant

The moral

- There are multiple ways to calculate some quantities
  - `fib`, `ifib`
  - `sum-integers`, `sum-int`, `(/ (* n (+ n 1)) 2)`
- There is no particular relationship between program size and runtime

The moral

- Multiple *algorithms* exist to compute many values, and each can have different costs:
  - different number of operations they require
  - different memory requirements
  - so some are (much) more efficient than others

Note: Memoization

- There is a way to *automatically* convert inefficient tree-recursive `fib` procedure into much more efficient linear recursive version
- Also works for many other procedures
- Basic idea: have way to memorize values already computed (*e.g.* `(fib 2)`) so don't have to recompute them all the time
- This process is called "memoization"
  - We'll see it later in the course

Next topic

- Quantifying the efficiency of processes
- But first: a movie clip!

Nigel Tufnel's problem

- He doesn't understand the way a function (the volume of his amplifier, V) *scales* with the size of its inputs:
  - p_k: the position of the knob (0-11)
  - pow: the power rating of the amplifier
- He thinks $V(p_k, pow) = k_1 * p_k$
- In reality: $V(p_k, pow) = k_1 * p_k * pow$
  (ignoring nonlinearities)

Question

- How do we *quantify* the resources (time, space) that our procedures require?
- How do these resource requirements *change* with the size of the inputs?

Time complexity

- What we care about most is
  - How does the *run time* grow with input size?
  - *e.g.*
    - `sum-integers`, `ifib` grew linearly
    - `fib` grew exponentially
  - How can we quantify this?

Being precise

- Recall:

```
(define (sum-integers n)
  (if (= n 0)
      0
      (+ n (sum-integers (- n 1)))))
```

- Time = $(n+1) * (T_{cmp} + T_{if} + 2*T_{add} + T_{call})$

Not being *too* precise…

- Time = $(n+1) * (T_{cmp} + T_{if} + 2*T_{add} + T_{call})$
- More simply: Time = $C_1 + n*C_2$
- C_1 and C_2 are constant factors
- How does the run time *scale* as `n` gets larger?

Doing the numbers

- Time = $C_1 + n*C_2$
- Let's say $C_1 = 1000$, $C_2 = 10$
- n = 0 time = 1000
- n = 1 time = 1010
- n = 10 time = 1100
- n = 1000 time = 11000
- n = 10^6 time = 10001000
- Conclusion: C_2 *dominates* C_1 for large **n**

Doing the numbers

- Time = $C_1 + n*C_2$
- Run time is *linear* in **`n`**
- For large **`n`**, C_2 is all that really counts
  - C_1 has little effect
- How do we express this mathematically?

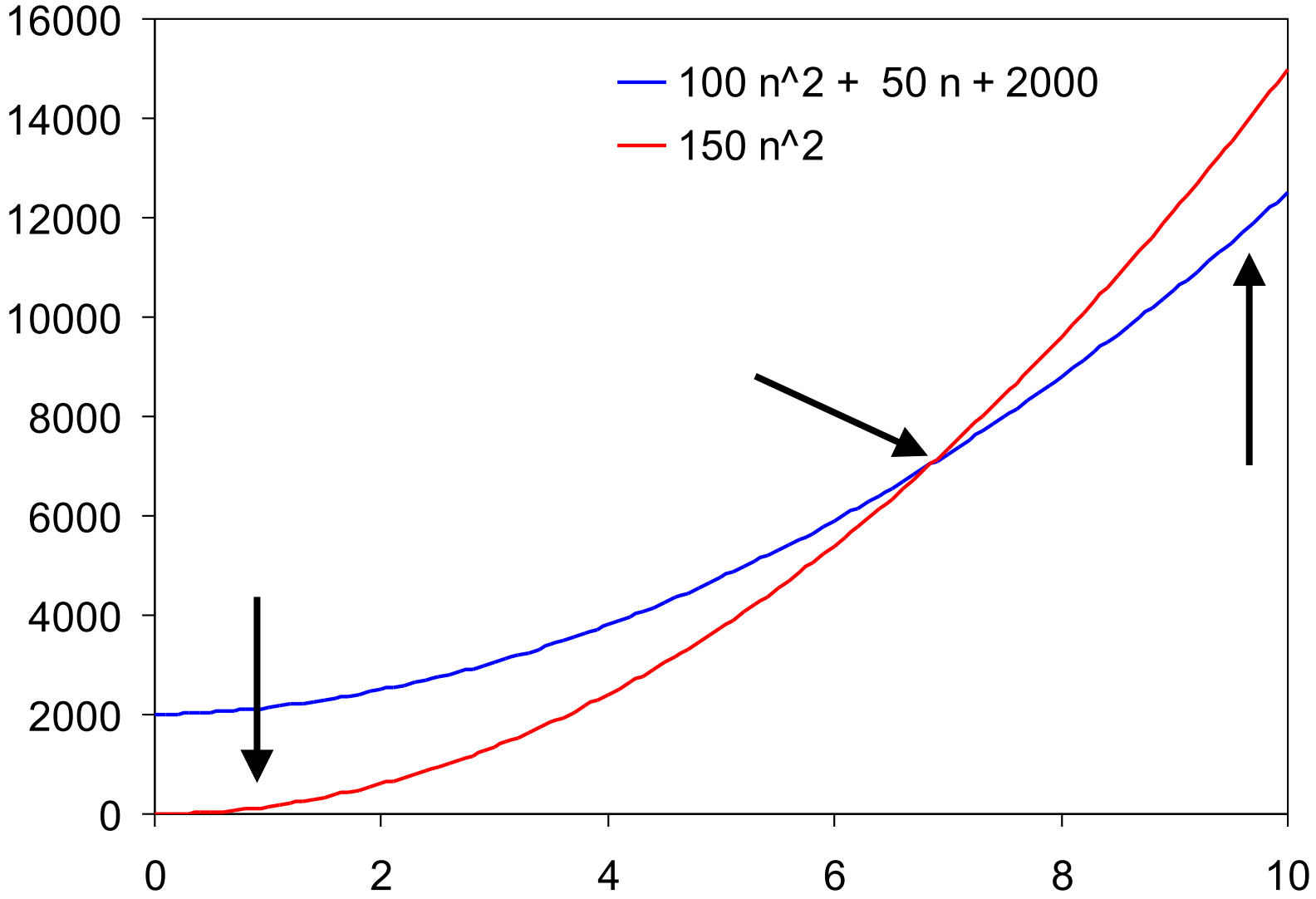
"Big O" notation

- A function g(n) is O(f(n)) if
  - for n greater than *some* n_0
  - then g(n) <= C * f(n)
    - for *some* constant C
- *i.e.* g(n) grows *no faster* than f(n) for sufficiently large n
- This is just math, not programming

Example of big O

- Let $g(n) = 100 * n^2 + 50 * n + 2000$
  $<= 150 * n^2$ for large enough n

- So $g(n)$ is $O(n^2)$
  - here, $C = 150$
  - note: $C = 100.00001$ would also work


16000
14000
12000
10000
8000
6000
4000
2000
0
0
2
4
6
8
10
100 n^2 + 50 n + 2000
150 n^2


CALIFORNIA INSTITUTE OF TECHNOLOGY
1891

A tighter bound…

- $g(n) = 100 * n^2 + 50 * n + 2000$

  $g(n)$ is $O(n^2)$

  but $g(n)$ is also $O(n^3)$!

  $g(n) <= n^3$ for large n

  $O(n^2)$ is not a "tight bound"
- Need another concept…

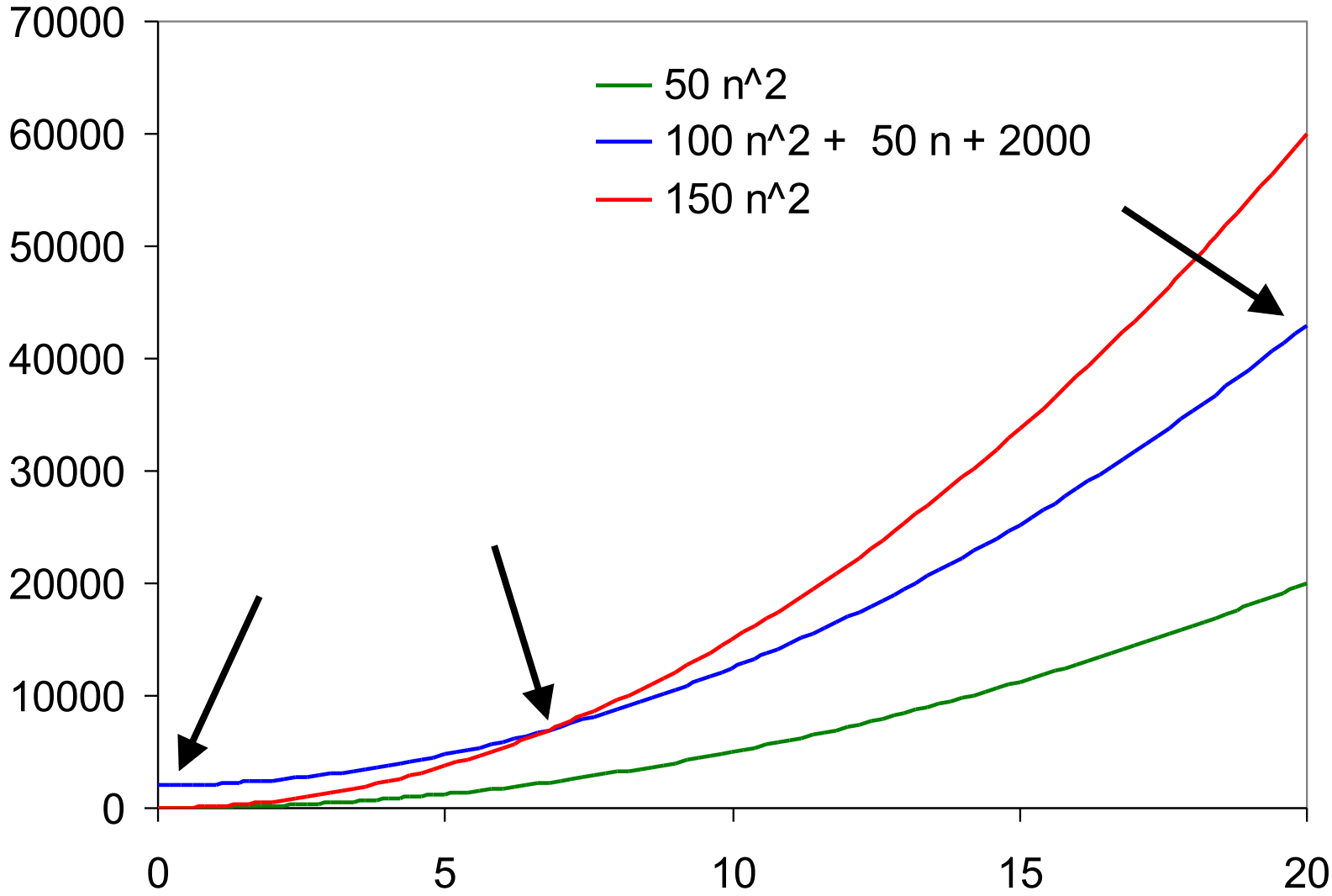
Big theta (Θ)

- $g(n) = 100 * n^2 + 50 * n + 2000$

  $g(n)$ is $\Theta(n^2)$
  - means: $C_1 * n^2 <= g(n) <= C_2 * n^2$
    - for sufficiently large n
    - for some C_1 and C_2 , $C_2 >= C_1$
  - *e.g.* $C_1 = 50$, $C_2 = 150$
  - this *is* a tight bound
  - $C_1 * n^2$ and $C_2 * n^2$ "sandwich" $g(n)$

Big theta (Θ)

- $g(n) = 100 * n^2 + 50 * n + 2000$
- $C_1 * n^2 <= g(n) <= C_2 * n^2$
  - Another way to think about it:
    - $g(n)$ grows *no faster* than n^2
    - $g(n)$ grows *no slower* than n^2
    - so $g(n)$ grows "as fast as" n^2
  - n^2 is a precise expression of scaling behavior of $g(n)$


70000
60000
50000
40000
30000
20000
10000
0
0
5
10
15
20
50 n^2
100 n^2 + 50 n + 2000
150 n^2


CALIFORNIA INSTITUTE OF TECHNOLOGY
1891

Abuse of notation

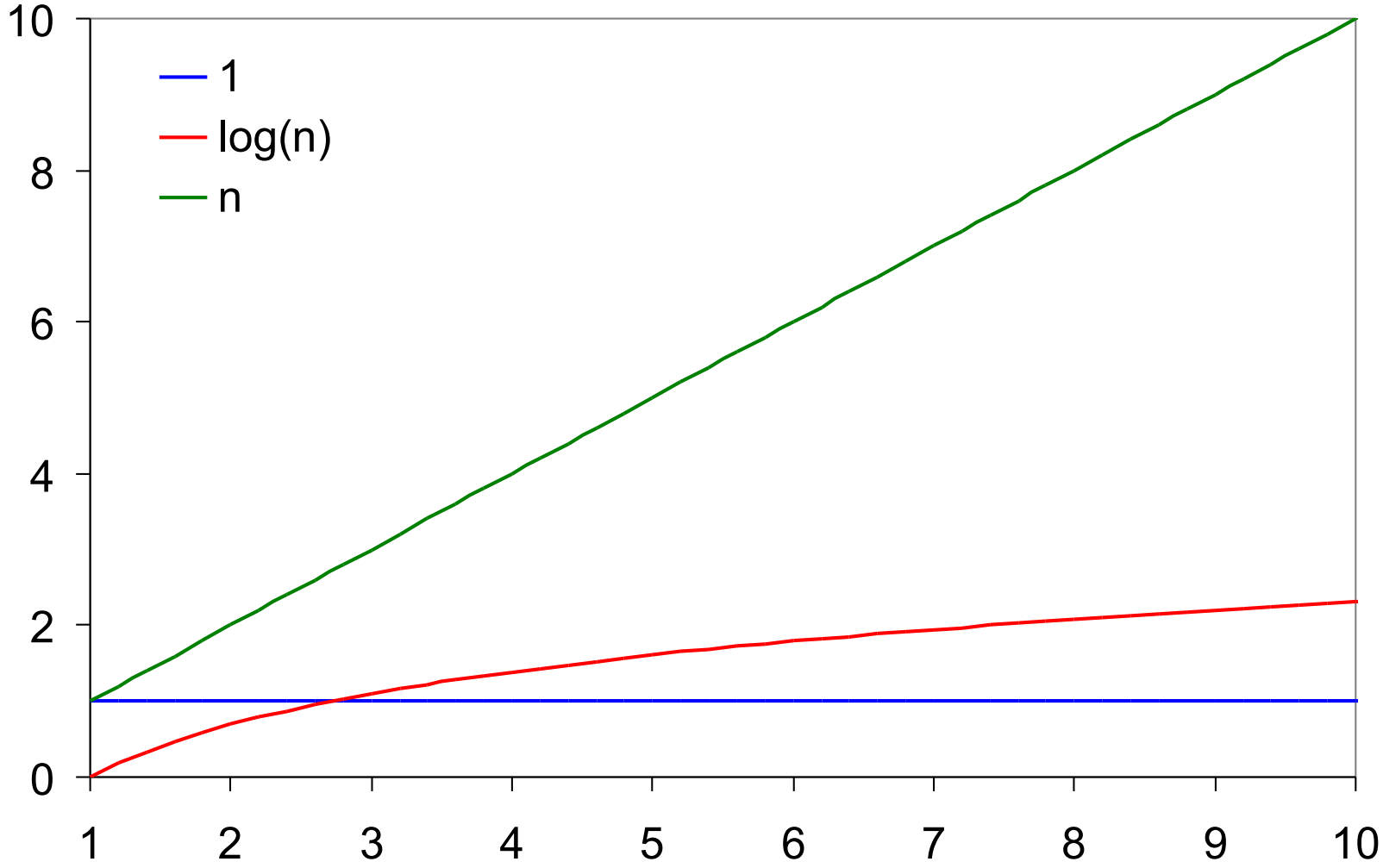
- When we say $O(n^2)$, we almost always really mean $\Theta(n^2)$
- We prefer to use the tightest bound
- "oh" is easier to say than "theta"
- However…
  - `fib` is $O(2^n)$ but $\Theta(g^n)$ where g is the "golden ratio" (1.618…)

Asymptotic Complexity

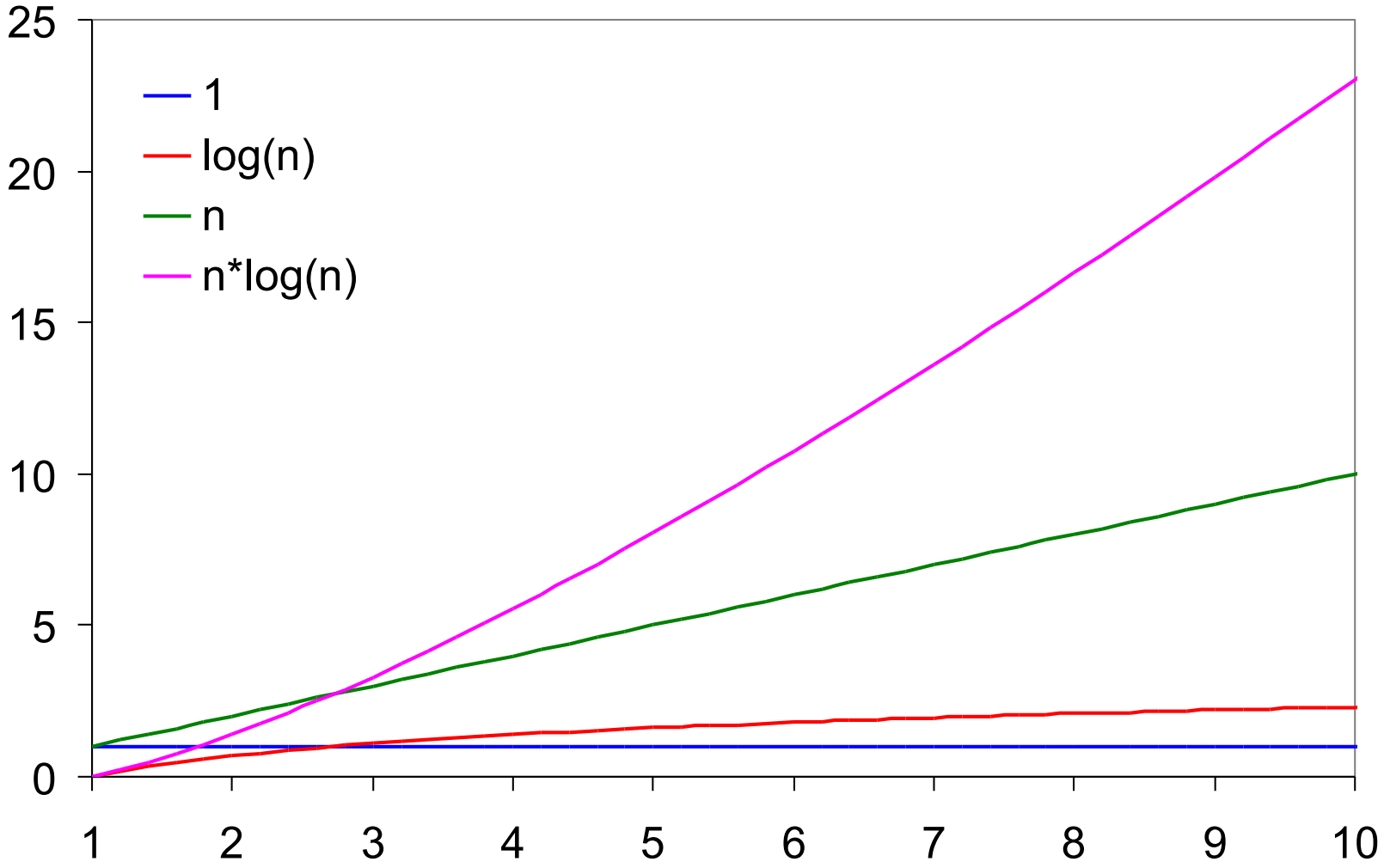
- Intuitively:
  - only care about fastest growing term
    - other terms don't matter for large n
  - don't care (much) about constant factors
    - they usually reflect the implementation more than the algorithm
- Usually compare to simple functions
  - *e.g.* 1, log(n), n, n*log(n), n^2, 2^n

Typical functions

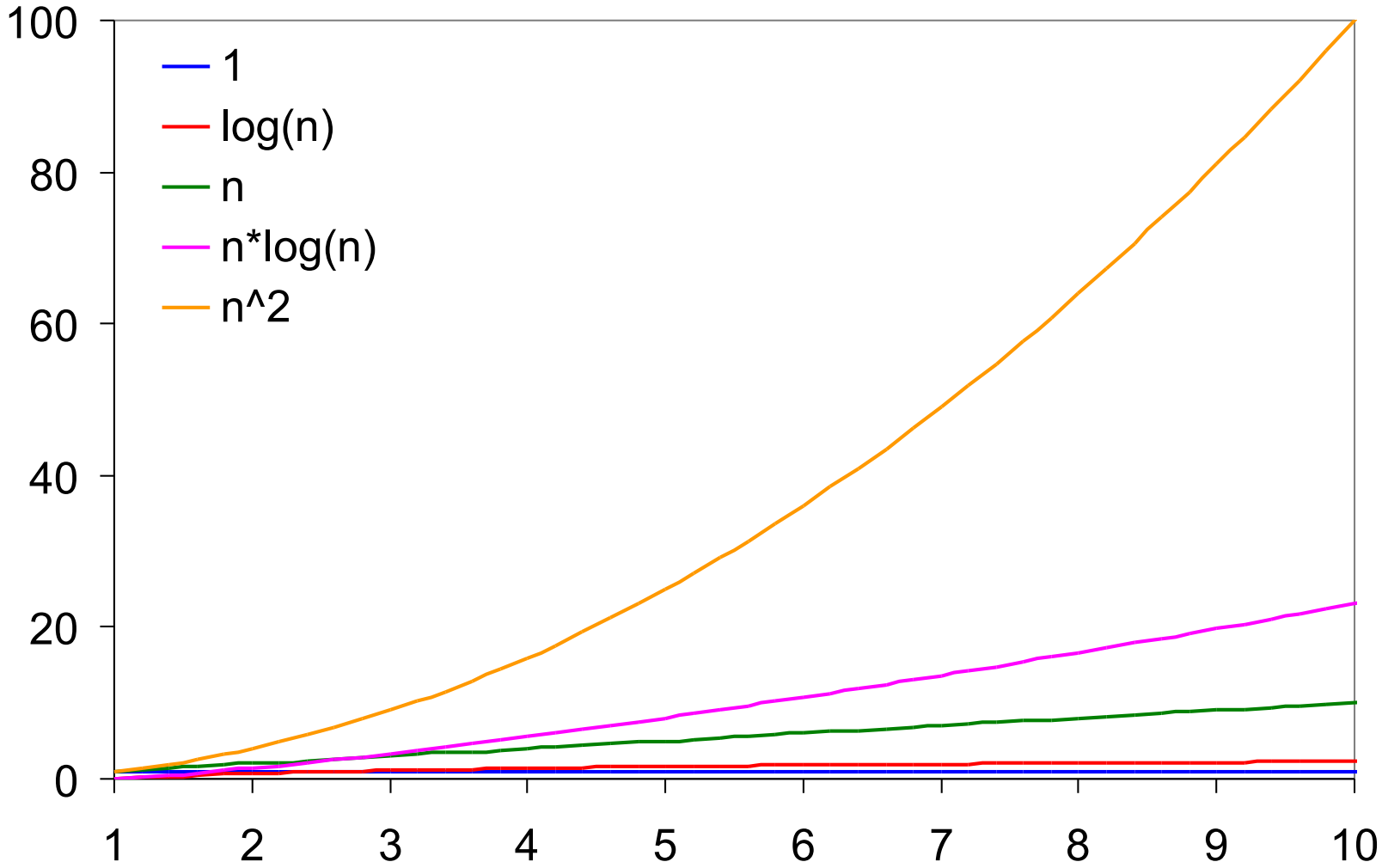
- 1 — constant
- log(n) — logarithmic
- n — linear
- n * log(n) — supralinear
- n^2 — quadratic
- 2^n — exponential


10
8
6
4
2
0
1
log(n)
n
1
2
3
4
5
6
7
8
9
10

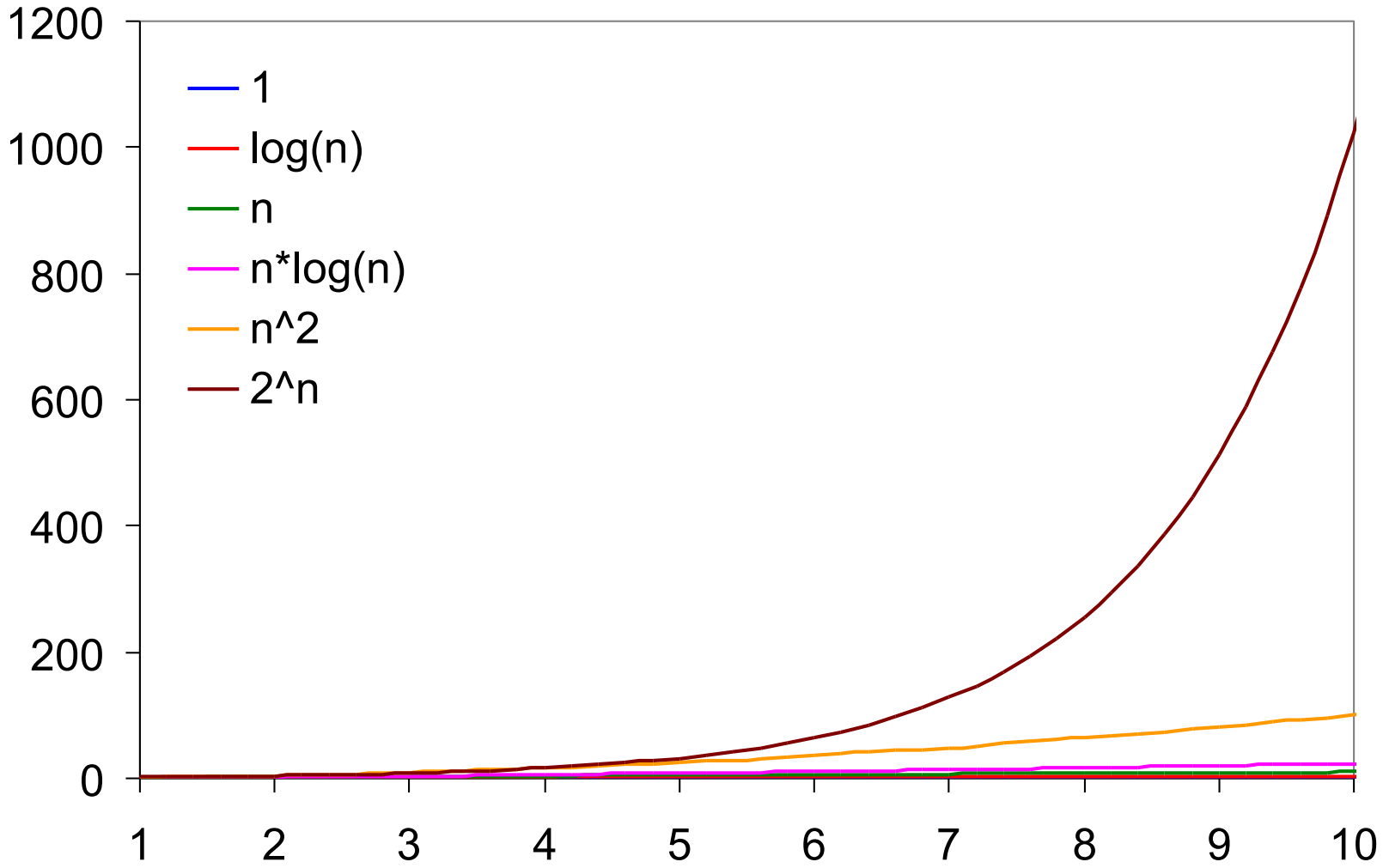

CALIFORNIA INSTITUTE OF TECHNOLOGY
1891


25
20
15
10
5
0
1
log(n)
n
n*log(n)
1
2
3
4
5
6
7
8
9
10


CALIFORNIA INSTITUTE OF TECHNOLOGY
1891


100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
1
log(n)
n
n*log(n)
n^2


CALIFORNIA INSTITUTE OF TECHNOLOGY
1891


1200
1000
800
600
400
200
0
1
log(n)
n
n*log(n)
n^2
2^n
1
2
3
4
5
6
7
8
9
10

Examples

- Any constant is O(1)

```
(define (fast-sum-integers n)
   (/ (* n (+ n 1)) 2))
```

- T(**`fast-sum-integers`**)

$= T_{div} + T_{add} + T_{mul} = \text{constant}$

$= C_1 * 1$

$= O(1)$

Examples

- T(**`sum-integers`**)

  $= (N+1) * (T_{cmp} + T_{if} + 2*T_{add} + T_{call})$

  $= C_1 + N*C_2$

  $= O(N)$ (linear)

Examples

- Recall from last lecture:

$$T(\text{fib}) <= 2^{(n+1)} * T_{fib}$$
$$= 2^{(n+1)} * C_1$$
$$= 2^n * (2*C_1) = 2^n * C_2$$
$$= O(2^n) \text{ (exponential)}$$

Why should we care?

- Recursive vs. iterative fibonacci function:
- Recursive:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)) )))
```

- Iterative:

```
(define (ifib n) (fib-iter 1 0 n))
(define (fib-iter fnext f cnt)
  (if (= cnt 0)
      f
      (fib-iter (+ fnext f) fnext (- cnt 1))))
```

Who cares?

- Recursive vs. iterative fibonacci function:
- Recursive: $O(2^n)$
- Iterative: $O(n)$
- n = 0, 1: no difference
- n = 10:
  - Recursive: ~1024 time units
  - Iterative: ~10 time units
- n = 1000: try it!
  - (might not want to wait for recursive function)

More examples

- Not all procedures have linear or exponential time complexity
- What about this one?

```
(define (power-of-two? n)
  (if (= n 1)
      #t
      (if (< n 1)
          #f
          (power-of-two? (/ n 2)))))
```

- What is its time complexity?

Intuitively…

- For a positive integer n, want to find if it's an integral power of two
  - *e.g.* 1, 2, 4, 8, 16…
- Do this by repeatedly dividing by 2:
  - If we hit 1 exactly, it's a power of two
  - if less than 1, it isn't
- *e.g.*

  8: 4, 2, 1 (ok!)

  7: 7/2, 7/4, 7/8 (nope!)

Evaluating…

- `(power-of-two? 8)`

```
(= 8 1) ; #f
(< 8 1) ; #f
(power-of-two? 4)
...
(power-of-two? 2)
(power-of-two? 1)
(= 1 1)
```

- Answer: `#t` (`8` is a power of two)

Continued…

- `(power-of-two? 7)`

```
(= 7 1) ; #f
(< 7 1) ; #f
(power-of-two?  7/2)
...
(power-of-two?  7/4)
(power-of-two?  7/8)
(= 7/8 1) ; #f
(< 7/8 1) ; #t
```

- Answer: **#f** (7 is not a power of two)

Continued…

```
(define (power-of-two? n)
  (if (= n 1)
      #t
      (if (< n 1)
          #f
          (power-of-two? (/ n 2)))))
```

- $T_{\texttt{p-o-t}} = 2*T_{if} + 2*T_{cmp} + T_{div} + T_{call} = C_1$
  - constant amount of work for each procedure call (as usual)
- How many calls to **`power-of-two?`**

Continued…

- n = 8
- n = 4
- n = 2
- n = 1
- n = 7

4 calls = $\log_2(8) + 1$

3 calls = $\log_2(4) + 1$

2 calls = $\log_2(2) + 1$

1 call = $\log_2(1) + 1$

also 4 calls =

ceil($\log_2(7)$) + 1

Continued…

- In general…

  #calls(n) = $\log_2(n) + 1$, rounded up

  T(`power-of-two?`) = $C_1 * (\log_2(n) + 1)$

  $= C_1 * \log_2(n) + C_1$

  $= O(\log(n))$

- "logarithmic" time complexity

A subtle point

- Why $O(\log(n))$ and not $O(\log_2(n))$?

  $a^x = n$

  $x = \log_a(n)$

  $\log_2(a^x) = x*\log_2(a) = \log_2(n)$

  $x = \log_2(n) / \log_2(a) = \log_a(n) = k * \log_2(n)$

- Logarithms with different bases are equivalent to within a constant factor (here, $k = 1/\log_2(a)$)
- We don't care about constant factors
- so we say $O(\log(n))$, not $O(\log_2(n))$

Pop quiz

```
(define (power-of-two? n)
  (if (= n 1)
      #t
      (if (< n 1)
          #f
          (power-of-two? (/ n 2)))))
```

- What is the *space* complexity?

Rules of thumb

- To determine time complexity, must consider
  - which arguments control termination of procedure
  - how fast they change from one call to the next
- When we *subtract* a fixed amount from the argument per recursive call
  - usually linear complexity
- When we *divide* the argument by a fixed amount per recursive call
  - usually logarithmic complexity

Rules of thumb

- However, this assumes only *one* recursive call per procedure
- With more, rules of thumb don't apply
  - *e.g.* `fib`

Another example

- Given a number n, can we find a and b less than n such that $a^2 + b^2 = n^2$ (a, b, n all integers > 0)?

```
(define (sum-of-squares? a b n)
  (= (+ (* a a) (* b b)) (* n n)))
```

- Time complexity of **`sum-of-squares?`**

  O(1) (but we're not done yet)

Simple algorithm

- Solve by checking all pairs (a, b)
  - where a < n and b < n
- Start with a = 1, b = 1
- If $a^2 + b^2 = n^2$, succeed
- else increment b
- When b >= n, reset b to 1, increment a
- When a >= n, fail!

Translate to Scheme...

```
(define (is-sum-of-squares? n)
  (iter 1 1 n)) ; start with a = 1, b = 1

(define (iter a b n) ; helper function
  (if (>= a n) #f
      (if (>= b n) (iter (+ a 1) 1 n)
          (if (sum-of-squares? a b n) #t
              (iter a (+ b 1) n)))))
```

Aside 1: internal procedures

- The helper function `iter` is only used inside the function `is-sum-of-squares?`
- It seems silly to dump this name into the global namespace when it's only used in one place
- To deal with this, Scheme lets you define procedures inside other procedures
- Such "internal procedures" can only be used inside the procedure in which they're defined

Aside 1: internal procedures

- This:

```
(define (is-sum-of-squares? n)
  (iter 1 1 n)) ; start with a = 1, b = 1

(define (iter a b n)
  (if (>= a n) #f
      (if (>= b n) (iter (+ a 1) 1 n)
          (if (sum-of-squares? a b n)  #t
              (iter a (+ b 1) n)))))
```

Aside 1: internal procedures

- Can be written like this:

```
(define (is-sum-of-squares? n)
  (define (iter a b n)
    (if (>= a n) #f
        (if (>= b n) (iter (+ a 1) 1 n)
            (if (sum-of-squares? a b n)  #t
                (iter a (+ b 1) n)))))
  (iter 1 1 n))
```

internal procedure

Aside 1: internal procedures

- Advantages:
  - easier to read and understand code
  - no worry about name clashes
    - can call internal procedure *e.g.* `iter` in every function if you want
  - doesn't leave local functions at top level
- Disadvantage:
  - harder to debug in most implementations

Aside 2: cond

- A new special form
- Makes the previous code much more readable
- No more long chains of ifs...

From `if`...

```
(define (is-sum-of-squares? n)
  (define (iter a b n)
    (if (>= a n)  #f
        (if (>= b n) (iter (+ a 1) 1 n)
            (if (sum-of-squares? a b n) #t
                (iter a (+ b 1) n)))))
  (iter 1 1 n))
```

... to cond

```
(define (is-sum-of-squares? n)
  (define (iter a b n)
    (cond ((>= a n) #f)
          ((>= b n) (iter (+ a 1) 1 n))
          ((sum-of-squares? a b n) #t)
          (else  (iter a (+ b 1) n))))
  (iter 1 1 n))
```

if vs cond

```
(if (>= a n) #f
    (if (>= b n) (iter (+ a 1) 1 n)
        (if (sum-of-squares? a b n)  #t
            (iter a (+ b 1) n))))
```

```
(cond ((>= a n) #f)
      ((>= b n) (iter (+ a 1) 1 n))
      ((sum-of-squares? a b n)  #t)
      (else  (iter a (+ b 1) n)))
```

`cond` evaluation

```
(cond (<test1> <expr1>)
      (<test2> <expr2>)
      (...)
      (else <expr>))
```

- `<test>`s are evaluated one at a time
  - if true, corresponding `<expr>` evaluated
  - if false, go to next `(<test> <expr>)` clause
- If none true, evaluate `<expr>` in else clause

`cond` evaluation

```
(cond (<test1> <expr1>)
      (<test2> <expr2>)
      (...)
      (else <expr>))
```

- Only one `<expr>` is evaluated
- Its value is value of entire `cond` expression

Back to asymptotic complexity...

Stepping through…

- `(is-sum-of-squares? 5)`
- `(iter 1 1 5)`
- `(iter 1 2 5)`
- …to `(iter 1 5 5)`, then…
- `(iter 2 1 5)`
- … `(iter 3 4 5)`

  Got it! $3^2 + 4^2 = 5^2$
- answer: `#t`

Stepping through…

- `(is-sum-of-squares? 6)`
- `(iter 1 1 6)`
- `(iter 1 2 6)`
- …to `(iter 1 6 6)`, then…
- `(iter 2 1 6)`
- … `(iter 3 1 6)`
- … `(iter 6 1 6)` `; nope!`
- answer: `#f`

Counting up…

- What is the worst-case number of calls to `iter`?
  - `(iter 1 1 6)` to `(iter 1 6 6)` [6]
  - `(iter 2 1 6)` to `(iter 2 6 6)` [6]
  - … `(iter 5 1 6)` to `(iter 5 6 6)` [6]
  - `(iter 6 1 6)` and then stop!
- 31 calls, or $6^2 - 6 + 1$
- In general, at most $n^2 - n + 1$ calls
- Constant amount of work per call
- Therefore, time complexity: $O(n^2)$

Summary

- Order of growth is the most important factor in determining run time
- Can quantify order of growth using big-O (big-Θ) notation
- Procedures can have many different orders of growth
- Reducing order of growth can have enormous effects on run time

Next time

- Higher-order functions!
  - a different way to abstract procedures using procedures as data